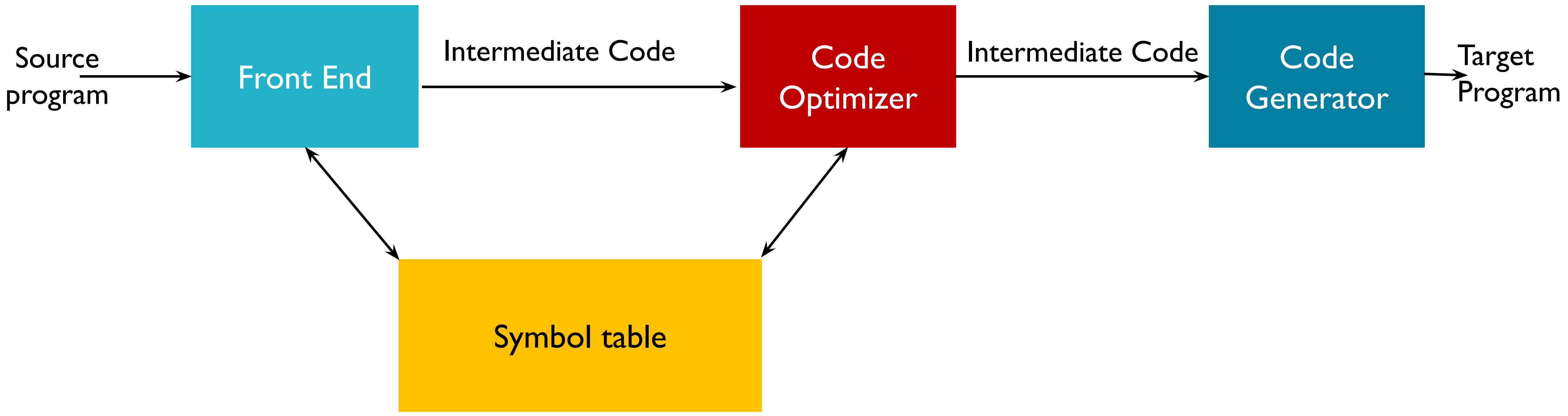
Code Generation & Code Optimization

Position of Code Generator



Issues in Design of a Code Generator

- Input to the code Generator
  - There are several choice for the intermediate language including
  - Linear representation such as postfix notation
  - Three-address representation such as quadruples
  - Virtual machine representation such as stack machine code
  - Graphical representation such as syntax tree and DAG
- Target Program
  - Absolute machine language program
  - Relocatable machine language program
  - Assembly language program
- Memory management

Issues in Design of a Code Generator

- Instruction Selection
- Register Allocation
- Choice of evaluation order

Basic Blocks in Flow Graph

- The basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching.
- Any given program can be partitioned into basic blocks by using following algorithm.
  - 1. First determine the leaders by using following rules.
    - First statement is a leader.
    - Any target statement of conditional or unconditional **goto** is a leader.
    - Any statement that immediately **follow a goto** or unconditional goto is a leader.
  - 2. The basic block is formed starting at the leader statement and ending just before the next leader statement appearing.

Basic Blocks in Flow Graph

```
prod = 0;
i = 1;
do
{
  prod = prod + a[i] * b[i];
  i=i+1;
}while(i<=10);
```

Three Address Code

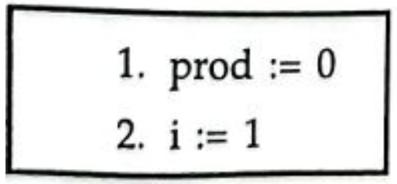
1. prod := 0
2. i := 1
3. t_1 := 4 * i
4. t_2 := a[t_1] /* computation of a[i] */
5. t_3 := 4 * i
6. t_4 := b[t_3] /* computation of b[i] */
7. t_5 := t_2 * t_4
8. t_6 := prod+t_5
9. prod := t_6
10. t_7 := i+1
11. i := t_7
12. if i< = 10 goto (3)

As per the algorithm,
Statement 1 is leader by rule 1(a)
Statement 3 is leader by rule 1(b)

Basic Blocks in Flow Graph

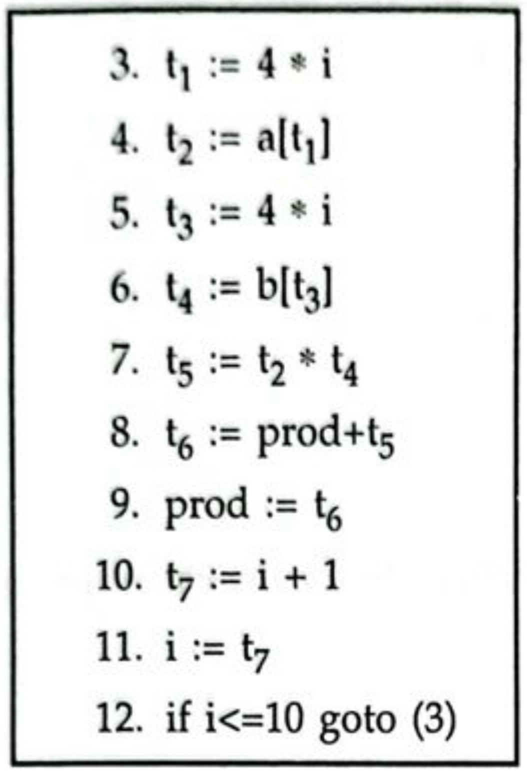
1. prod := 0
2. i := 1
3. t_1 := 4 * i
4. t_2 := a[t_1] /* computation of a[i] */
5. t_3 := 4 * i
6. t_4 := b[t_3] /* computation of b[i] */
7. t_5 := t_2 * t_4
8. t_6 := prod+t_5
9. prod := t_6
10. t_7 := i+1
11. i := t_7
12. if i< = 10 goto (3)

Block B1



1. prod := 0
2. i := 1

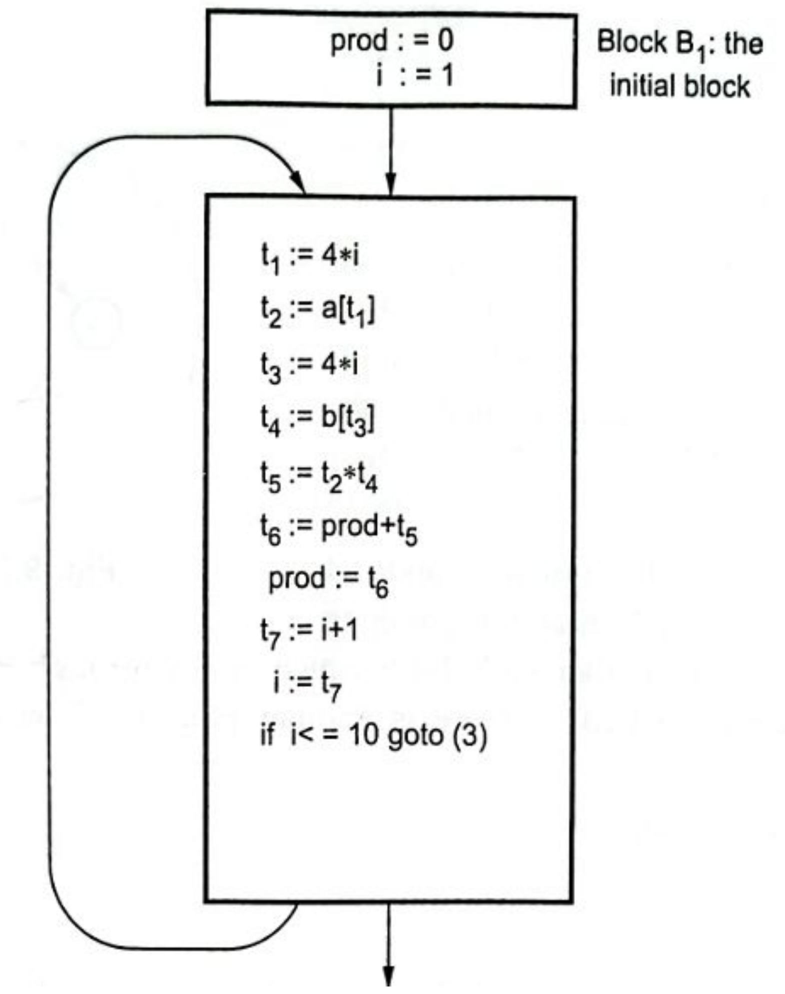
Block B2



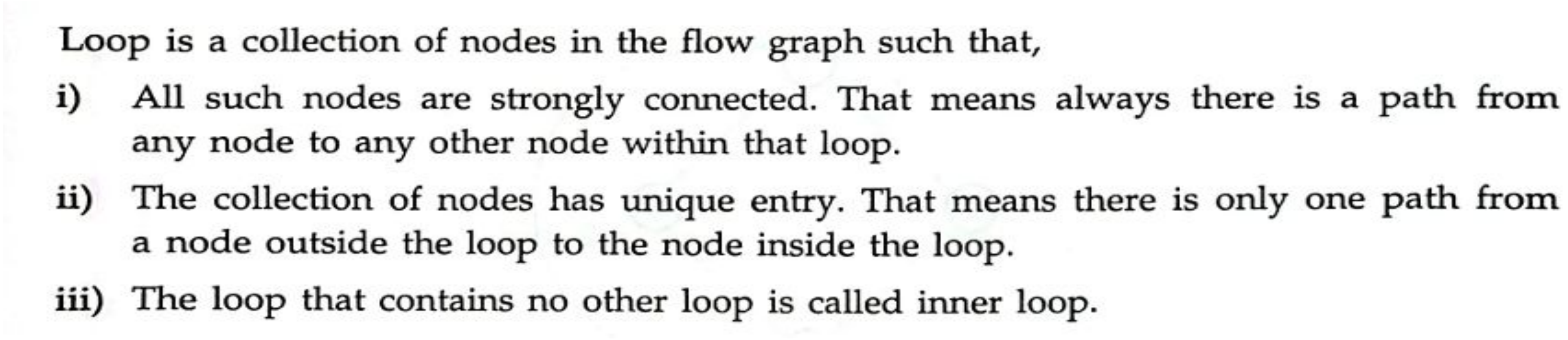
3. t_1 := 4 * i
4. t_2 := a[t_1]
5. t_3 := 4 * i
6. t_4 := b[t_3]
7. t_5 := t_2 * t_4
8. t_6 := prod+t_5
9. prod := t_6
10. t_7 := i + 1
11. i := t_7
12. if i<=10 goto (3)

Flow Graph

- Flow Graph is a directed graph in which the flow control information is added to the basic blocks.
  - The Nodes to the flow graph are represented by basic blocks.
  - The block whose leader is the first statement is called initial block.



Loops in Flow Graph



Loop is a collection of nodes in the flow graph such that,

i) All such nodes are strongly connected. That means always there is a path from any node to any other node within that loop.

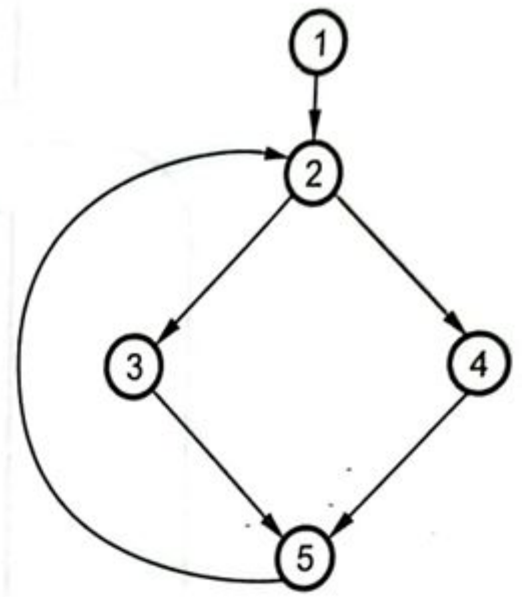
ii) The collection of nodes has unique entry. That means there is only one path from a node outside the loop to the node inside the loop.

iii) The loop that contains no other loop is called inner loop.

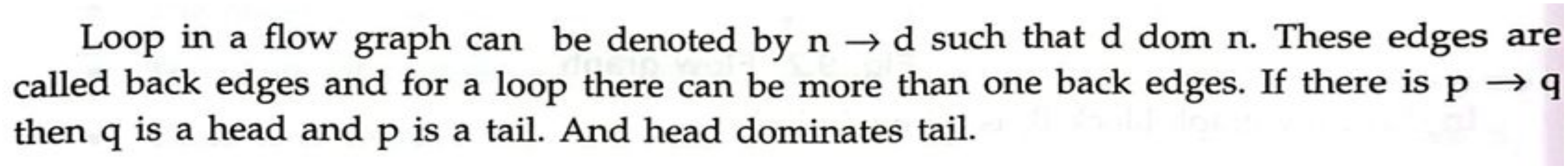
Dominator

- Dominators: A node d dominated n if every path to node n from initial node goes through d only. This can be denoted as ‘ d dom n’.
- Every initial node dominated all the remaining node in the flow graph.
- Every node dominated itself.

- Node 1 dominated every other node as it is initial node.
- Node 2 dominated 3, 4 and 5.
- Node 5 dominated no node.

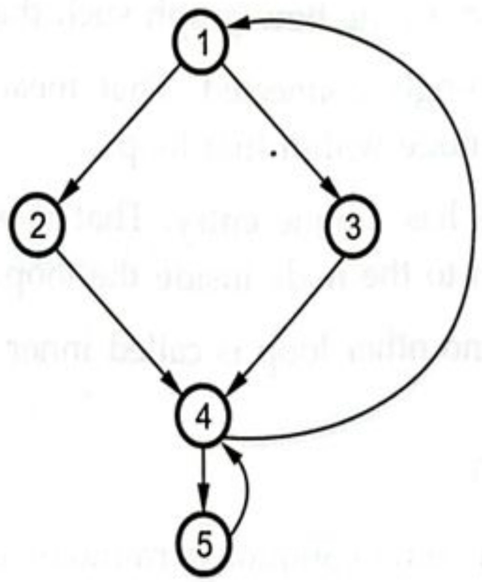


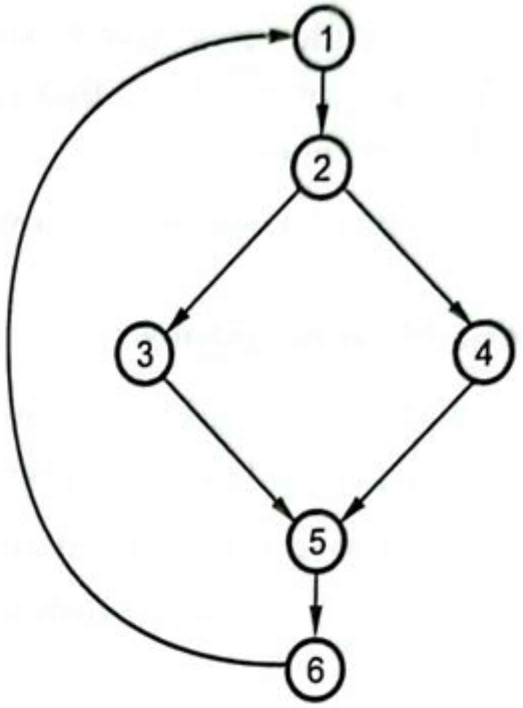
Natural Loop



Loop in a flow graph can be denoted by $n \rightarrow d$ such that d dom n. These edges are called back edges and for a loop there can be more than one back edges. If there is $p \rightarrow q$ then q is a head and p is a tail. And head dominates tail.

- The loops in the graph can be denoted by
  - 4 □ 1 (1 dom 4) and 5 □ 4 (4 dom 5)

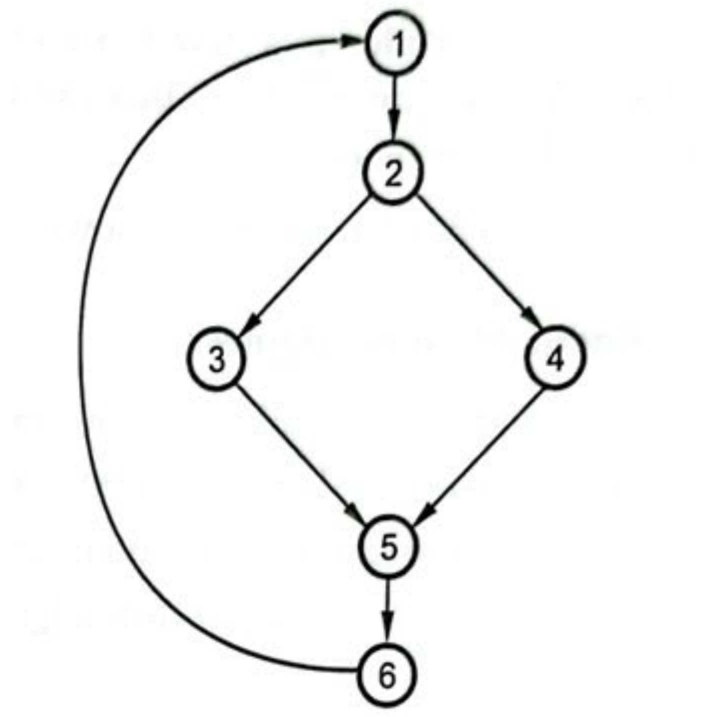




Natural Loop

- The natural loop can be defined by a back edge n→ d such that there exists a collection of all nodes that can reach to n without going through d and at the time d also can be added to this collection.
- 6 → 1 is natural loop because we can reach to all remaining

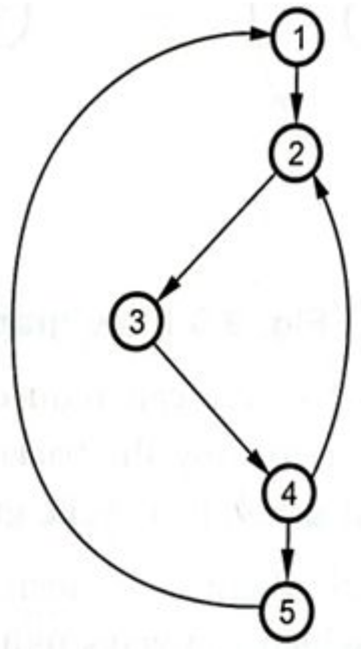
nodes from 6.



$$\begin{tikzcd}
 & 1 \arrow[d] & \\
 & 2 \arrow[dl] \arrow[dr] & \\
3 \arrow[dr] & & 4 \arrow[dl] \\
 & 5 \arrow[d] & \\
 & 6 \arrow[uuuu, bend left=80] &
\end{tikzcd}$$

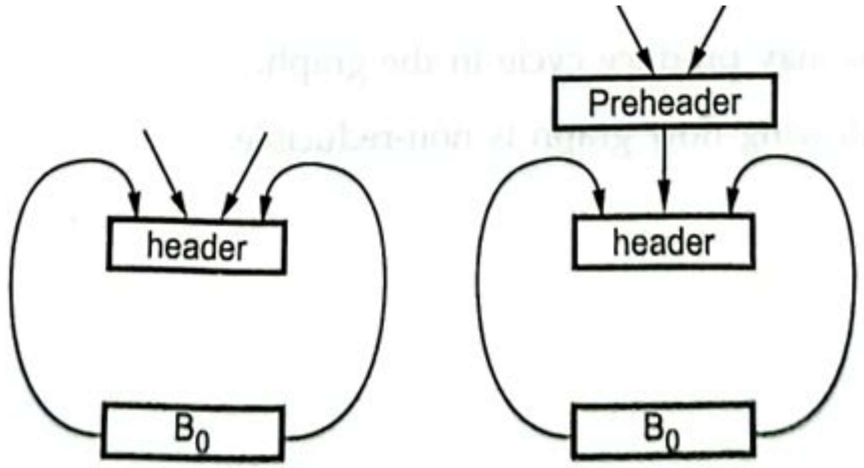
Inner Loop

- The inner loop is a loop that contains no other loop.
- 4 □ 2 is inner loop in the given graph.



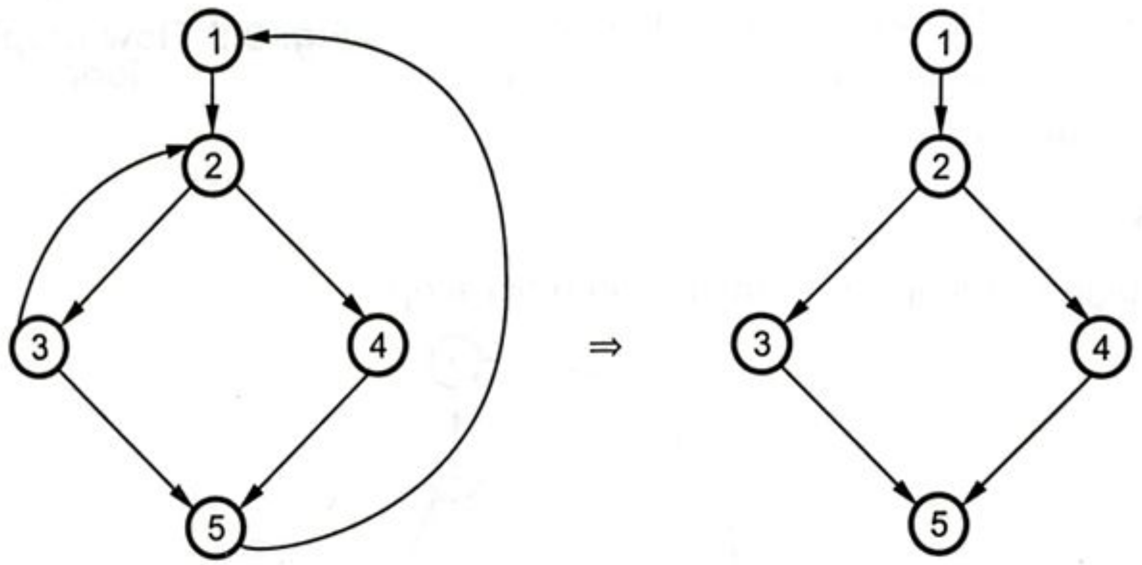
Pre Header

- The pre-header is a new block created such that successor of this block is the header block. All the computations that can be made before the header block can be made before pre-header block.



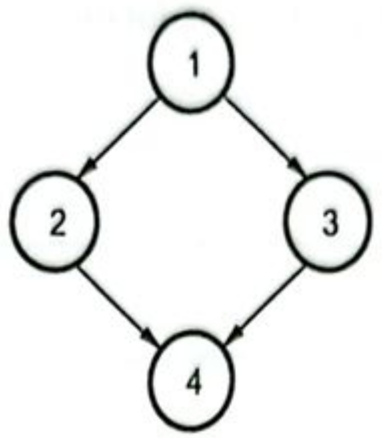
Reducible Flow Graph

- The reducible graph is a flow graph in which there are two types of edges, forward edges and back edges, with following properties:
  - The forward edge form an acyclic graph in which every node can be reached from an initial node
  - The back edges are such edges whose head dominated their tail.



Non-reducible Flow Graph

- The non-reducible flow graph is a flow graph in which
  - There are no back edges
  - The forward edge may contain cycle in the graph.

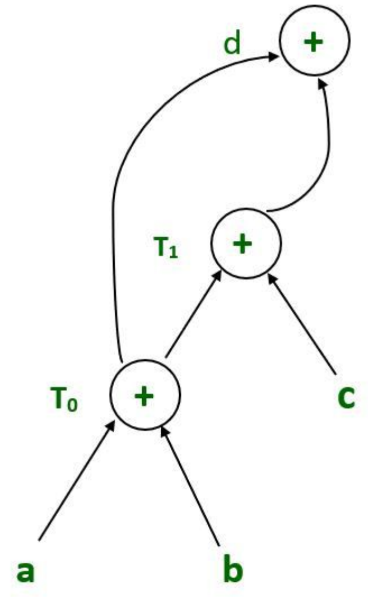


The DAG representation of basic blocks

- The Directed Acyclic Graph is a directed graph which contains no cycle.
  - DAG is used to optimize basic blocks
  - DAG is used to eliminate common subexpression
  - DAG specify how the value computed by each statement in a basic block is used in subsequent statement of the block.
  - To apply transformation on basic block, a DAG is constructed from three address code.
  - **Algorithm for construction of DAG:**
    - In a DAG, leaf node represents identifiers, variable names or constants
    - Interior node represents operators
    - While constructing DAG, a check is made to find if there is an existing node with the same children.

The DAG representation of basic blocks

- Construct DAG for the given expression:
  - (a+b)+(a+b+c)
  - Three address code:
    - t0 = a + b
    - t1 = t0 + c
    - d = t0 + t1



The DAG representation of basic blocks

Problem Statement

```
sum = 0;
for (i-0;i<=10;i++)
sum = sum+a[i];
```

Three Address Code

(1) sum :=0

(2) i;=0

(3) t_1 := 4*i

(4) t_2 := a[t_1]

(5) t_3 := sum+t_2

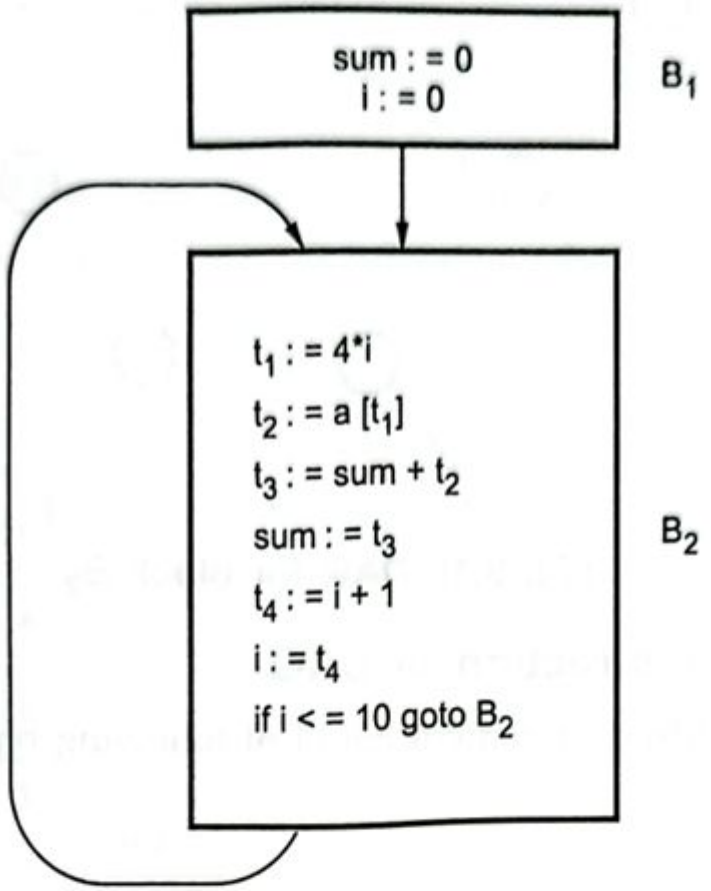
(6) sum := t_3

(7) t_4 = i+1;

(8) i:= t_4

(9) if i<=10 goto (3)

Flow Graph



The DAG representation of basic blocks

(1) $t_1 := 4 * i$

(2) $t_2 := a[t_1]$

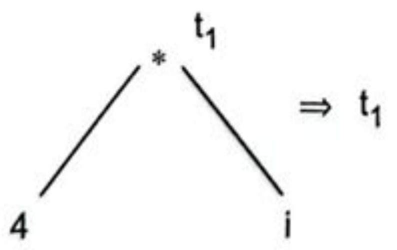
(3) $t_3 := sum + t_2$

(4) $sum := t_3$

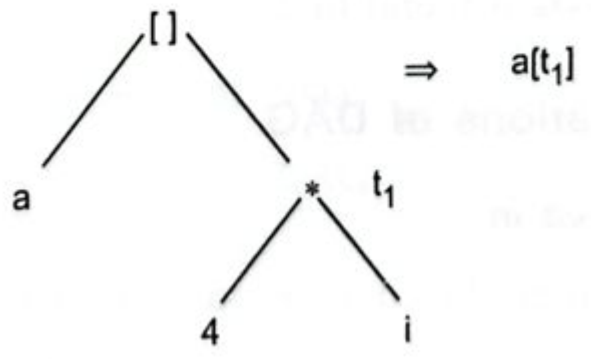
(5) $t_4 := i + 1$

(6) $i := t_4$

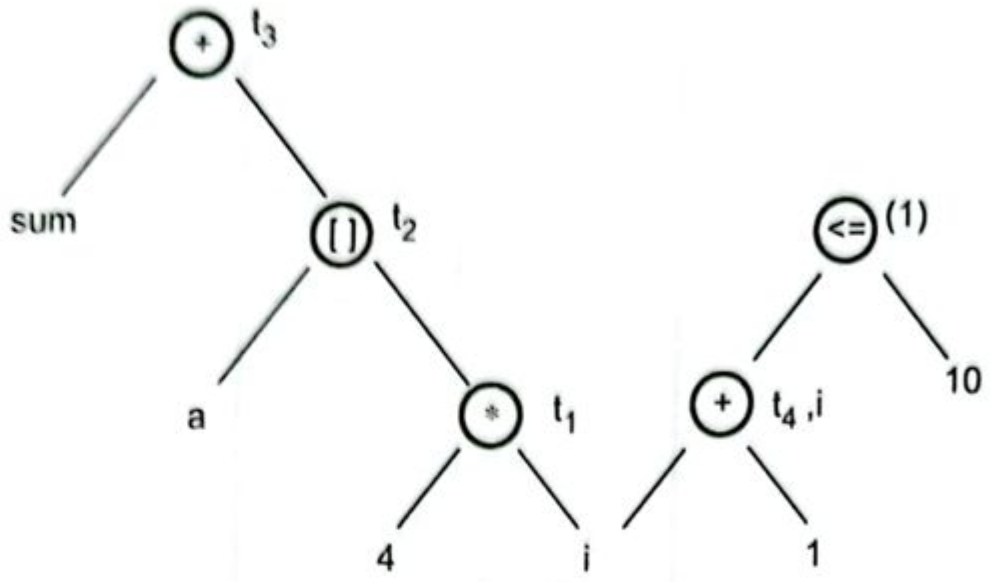
(7) if $i<10$ goto (1)



(a)



(b)



(c)

Code Optimization

- It mainly aims at rearranging the computation in a program so as to gain the advantage of execution speed, without changing the meaning of a program.
- It is necessary as
  - We want to utilize CPU efficiently
  - Efficient Power Consumption

Code Optimization

- **Platform(Machine) Dependent:**
  - Peephole optimization,
  - Instruction level parallelism,
  - Data level parallelism (keep data on many resources),
  - Cash optimization,
  - Redundant Resource
- **Platform Independent:**
  - Loop optimization,
  - Constant Folding,
  - Constant Propagation,
  - Common subexpression elimination,
  - Code Movement,
  - Dead code Elimination,
  - Strength Reduction

Peephole Optimization

- **Peephole optimization** is a type of code optimization performed on a small part of the code.
- It is performed on a very small set of instructions in a segment of code.
- It is a simple and effective technique for locally improving target code.
- **Characteristic of Peephole Optimization:**
    - Redundant Instruction Optimization
    - Flow of control optimization
    - Algebraic Simplification
    - Use of machine idioms

Redundant Instruction Optimization

Redundant Loads and Stores

If we see the instruction sequence
(1) MOV R0, a
(2) MOV a, R0
We can delete instruction (2) because whenever (2) is executed,
(1) will ensure that the value of a is already in register R0.

Unreachable Code

```
sum=0;
If(sum)
    printf("%d",sum);
```

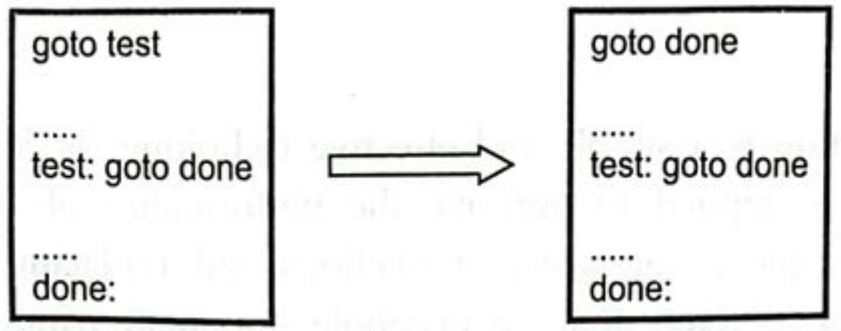
Last two statement can be eliminated

Redundant Loads and Stores

```
Initial code:
y = x + 5;
i = y;
z = i;
w = z * 3;

Optimized code:
y = x + 5;
w = y * 3; //* there is no i now
```

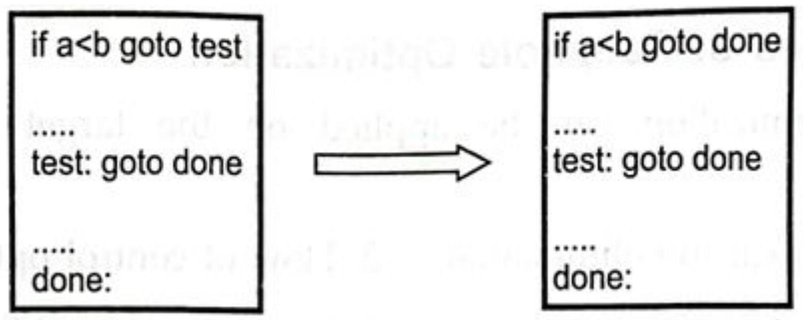
Flow-of-Control Optimization

Unnecessary jumps can be eliminated.



```
goto test

.....
test: goto done

.....
done:
```

```
goto done

.....
test: goto done

.....
done:
```



```
if a<b goto test

.....
test: goto done

.....
done:
```

```
if a<b goto done

.....
test: goto done

.....
done:
```

Algebraic Simplification

The statement such as

x = x + 0

or

x = x * 1

can be eliminated.

Use of machine idioms

- The target instructions have equivalent machine instructions for performing some operations. Hence we can replace these target instructions by equivalent machine instructions in order to improve the efficiency.
- For example,
  - Some machines have auto-increment or auto-decrement addressing modes that are used to perform add or subtract operations.

Reduction in Strength

- Certain machine instructions are cheaper than others. In order to improve the performance of the intermediate code, we can replace these instructions with equivalent but cheaper instructions.
- For example, squaring a number is cheaper than multiplying it by itself.
- Similarly, addition and subtraction are cheaper than multiplication and division. Therefore, we can effectively use equivalent addition and subtraction operations in place of multiplication and division to optimize performance.

Function Preserving Transformation

- Copy Propagation,
- Common subexpression elimination,
- Dead code Elimination,
- Constant Folding

C code for Quick Sort and Three Address Code

```
void quicksort(m,n)
int m,n;
{
     int i,j;
     int v,x;
     if ( n <= m ) return;
     /* fragment begins here */
     i = m-1; j = n; v = a[n];
     while(1) {
         do i = i+1; while ( a[i] < v );
         do j = j-1; while ( a[j] > v );
         if (i >= j ) break;
         x = a[i]; a[i] = a[j]; a[j] = x;
     }
     x = a[i]; a[i] = a[n]; a[n] = x;
     /* fragment ends here */
     quicksort(m,j); quicksort(i+1,n);
}
```

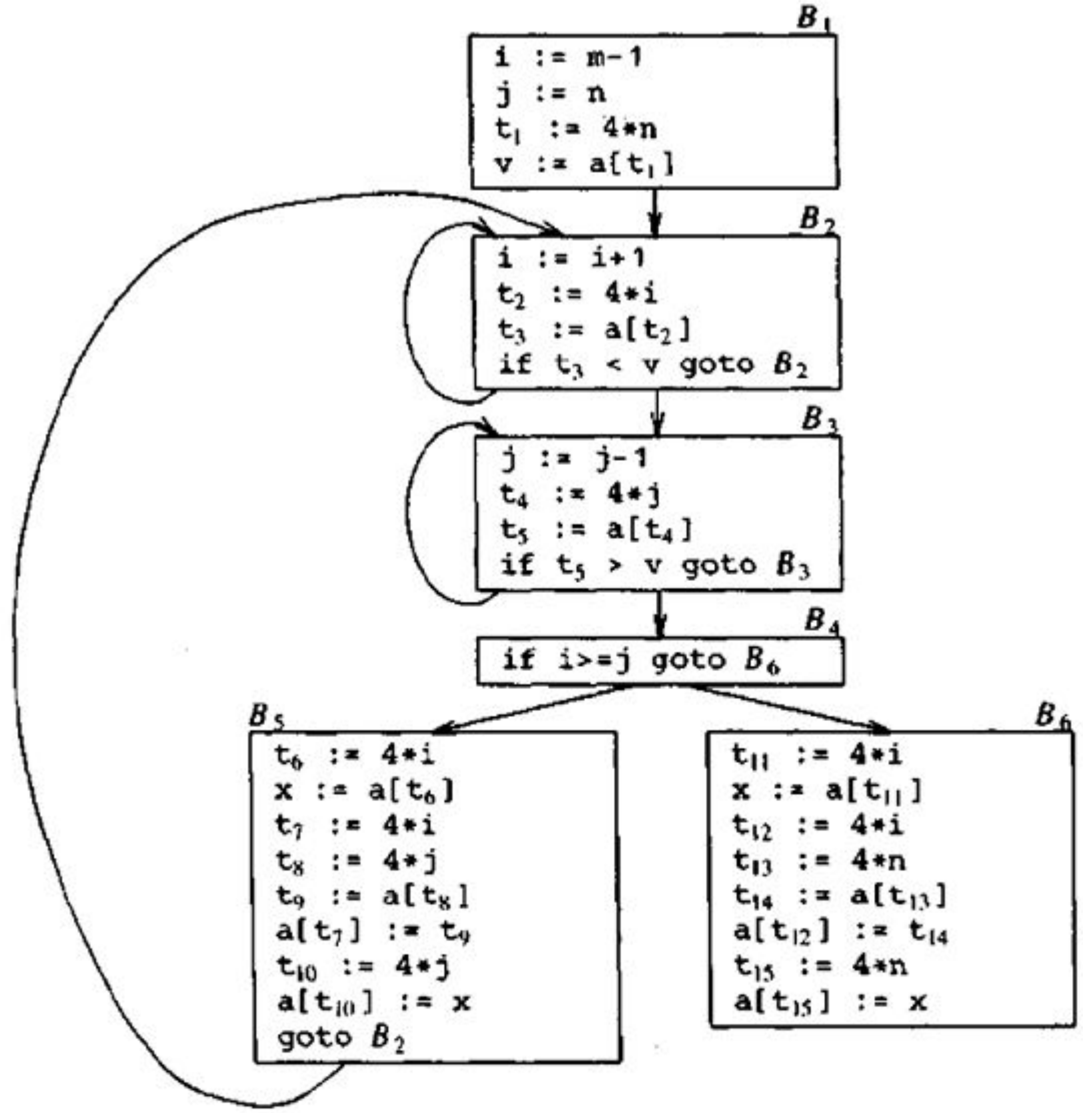
Fig. 10.2. C code for quicksort.

```
(1)    i := m-1
(2)    j := n
(3)   t1 := 4*n
(4)    v := a[t1]
(5)    i := i+1
(6)   t2 := 4*i
(7)   t3 := a[t2]
(8)   if t3 < v goto (5)
(9)    j := j-1
(10)  t4 := 4*j
(11)  t5 := a[t4]
(12)  if t5 > v goto (9)
(13)  if i >= j goto (23)
(14)  t6 := 4*i
(15)   x := a[t6]
```

```
(16)      t7 := 4*i
(17)      t8 := 4*j
(18)      t9 := a[t8]
(19)  a[t7] := t9
(20)     t10 := 4*j
(21) a[t10] := x
(22)      goto (5)
(23)     t11 := 4*i
(24)       x := a[t11]
(25)     t12 := 4*i
(26)     t13 := 4*n
(27)     t14 := a[t13]
(28) a[t12] := t14
(29)     t15 := 4*n
(30) a[t15] := x
```

Fig. 10.4. Three-address code for fragment in Fig. 10.2.

Flow Graph of Quick Sort



```
B1:
i := m-1
j := n
t1 := 4*n
v := a[t1]
```

```
B2:
i := i+1
t2 := 4*i
t3 := a[t2]
if t3 < v goto B2
```

```
B3:
j := j-1
t4 := 4*j
t5 := a[t4]
if t5 > v goto B3
```

```
B4:
if i>=j goto B6
```

```
B5:
t6 := 4*i
x := a[t6]
t7 := 4*i
t8 := 4*j
t9 := a[t8]
a[t7] := t9
t10 := 4*j
a[t10] := x
goto B2
```

```
B6:
t11 := 4*i
x := a[t11]
t12 := 4*i
t13 := 4*n
t14 := a[t13]
a[t12] := t14
t15 := 4*n
a[t15] := x
```

Fig. 10.5. Flow graph.

Local Sub Expression Elimination

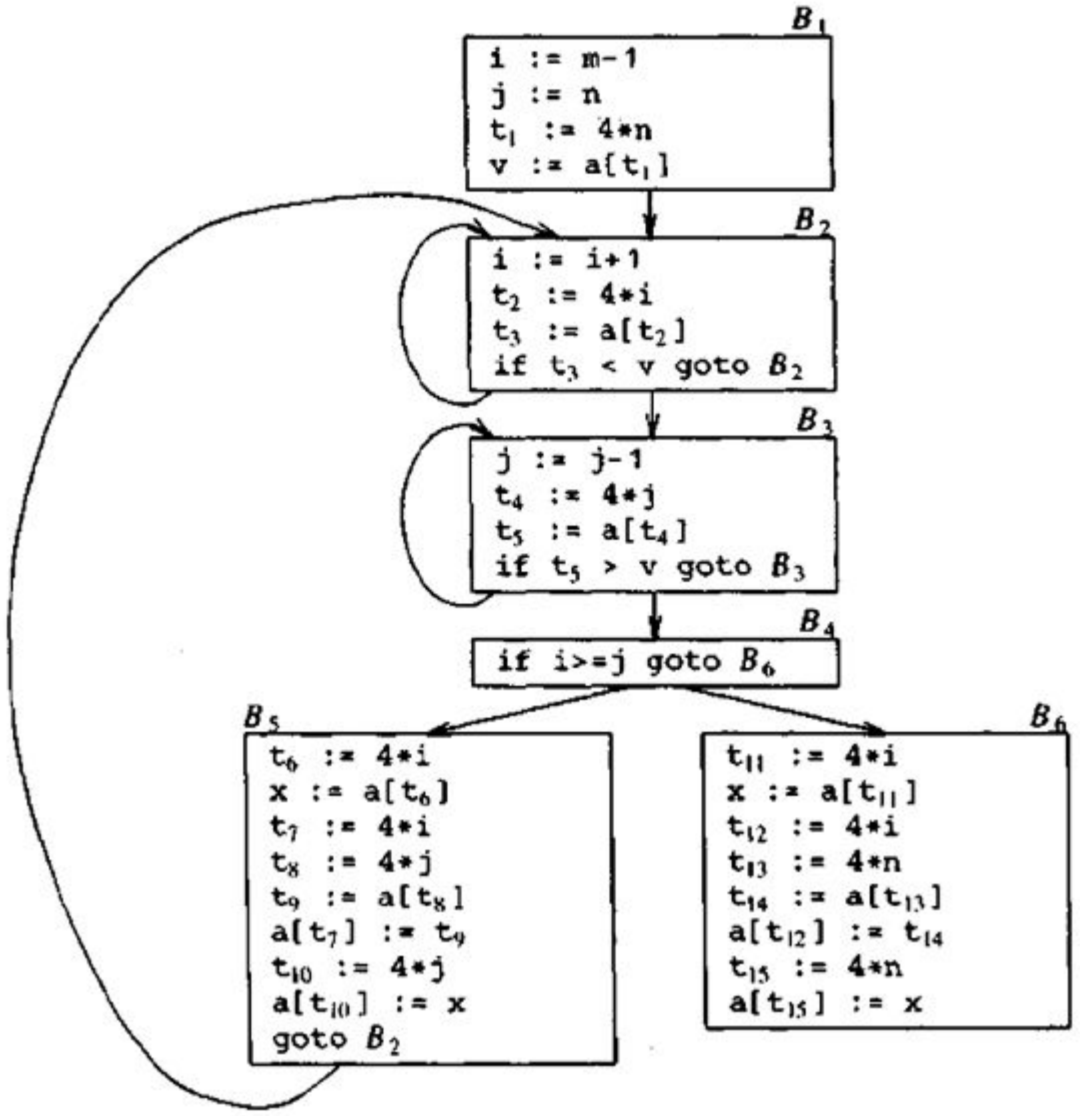
B5
```
t6  := 4*i
x   := a[t6]
t7  := 4*i
t8  := 4*j
t9  := a[t8]
a[t7] := t9
t10 := 4*j
a[t10] := x
goto B2
```

(a) Before

B5
```
t6  := 4*i
x   := a[t6]
t8  := 4*j
t9  := a[t8]
a[t6] := t9
a[t8] := x
goto B2
```

(b) After

Fig. 10.6. Local common subexpression elimination.



B1
```
i  := m-1
j  := n
t1 := 4*n
v  := a[t1]
```

B2
```
i  := i+1
t2 := 4*i
t3 := a[t2]
if t3 < v goto B2
```

B3
```
j  := j-1
t4 := 4*j
t5 := a[t4]
if t5 > v goto B3
```

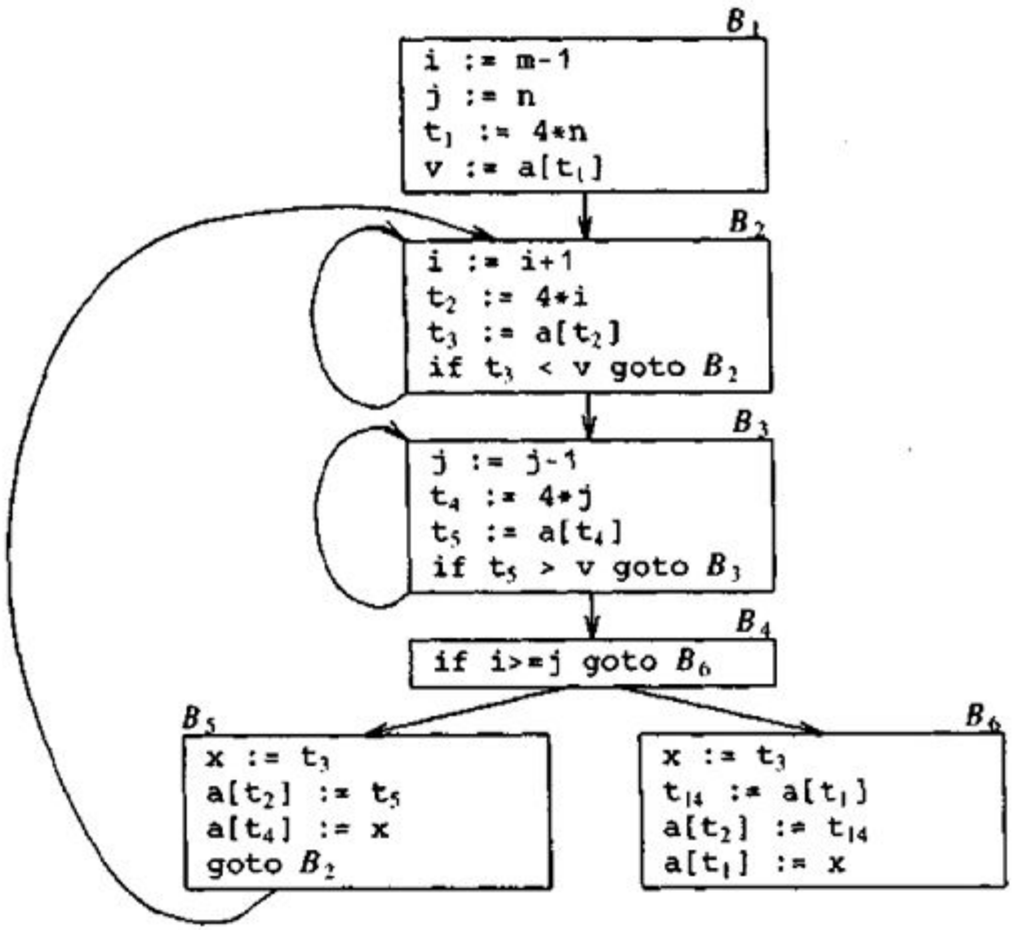
B4
```
if i>=j goto B6
```

B5
```
t6  := 4*i
x   := a[t6]
t7  := 4*i
t8  := 4*j
t9  := a[t8]
a[t7] := t9
t10 := 4*j
a[t10] := x
goto B2
```

B6
```
t11 := 4*i
x   := a[t11]
t12 := 4*i
t13 := 4*n
t14 := a[t13]
a[t12] := t14
t15 := 4*n
a[t15] := x
```

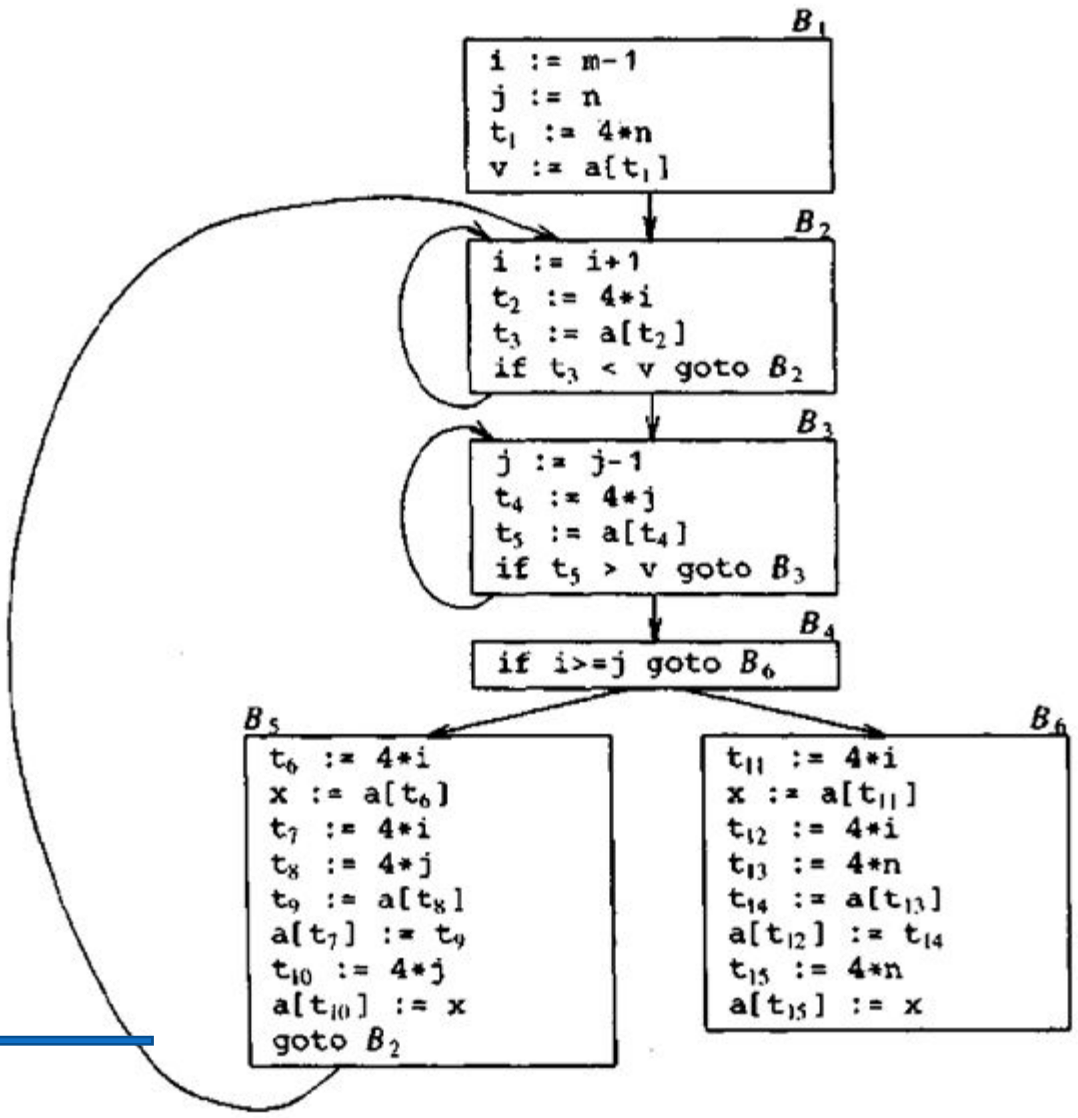
Fig. 10.5. Flow graph.

Common Sub Expression Elimination



```
B1
i := m-1
j := n
t1 := 4*n
v := a[t1]

B2
i := i+1
t2 := 4*i
t3 := a[t2]
if t3 < v goto B2

B3
j := j-1
t4 := 4*j
t5 := a[t4]
if t5 > v goto B3

B4
if i>=j goto B6

B5
x := t3
a[t2] := t5
a[t4] := x
goto B2

B6
x := t3
t14 := a[t1]
a[t2] := t14
a[t1] := x
```

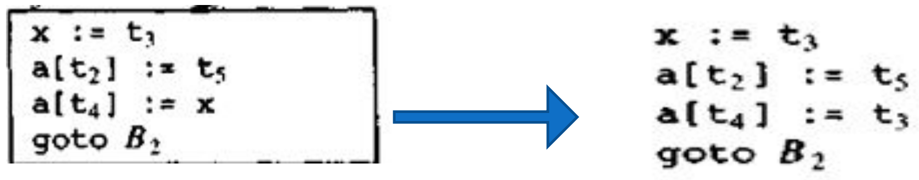
Fig. 10.7. B_5 and B_6 after common subexpression elimination.



```
B1
i := m-1
j := n
t1 := 4*n
v := a[t1]

B2
i := i+1
t2 := 4*i
t3 := a[t2]
if t3 < v goto B2

B3
j := j-1
t4 := 4*j
t5 := a[t4]
if t5 > v goto B3

B4
if i>=j goto B6

B5
t6 := 4*i
x := a[t6]
t7 := 4*i
t8 := 4*j
t9 := a[t8]
a[t7] := t9
t10 := 4*j
a[t10] := x
goto B2

B6
t11 := 4*i
x := a[t11]
t12 := 4*i
t13 := 4*n
t14 := a[t13]
a[t12] := t14
t15 := 4*n
a[t15] := x
```

Fig. 10.5. Flow graph.

Copy Propagation



```
x := t_3
a[t_2] := t_5
a[t_4] := x
goto B_2
```

```
x := t_3
a[t_2] := t_5
a[t_4] := t_3
goto B_2
```

```
//Before Optimization
    c = a * b
    x = a
    till
    d = x * b + 4


    //After Optimization
    d = a * b + 4
```

Constant Folding

Consider an expression : a = b op c and the values b and c are constants, then the value of a can be computed at compile time.

```
#define k 5
x = 2 * k
y = k + 5
```

This can be computed at compile time and the values of x and y are :

```
x = 10
y = 10
```

The value of PI can be replaced with 3.14 or √2 = 1.41 at compile time

Constant Propagation

If the value of a variable is a constant, then replace the variable with the constant.

The variable may not always be a constant.

(i) A = 2*(22.0/7.0)*r

Performs 2*(22.0/7.0)*r at compile time.

(ii) x = 12.4

y = x/2.3

Evaluates x/2.3 as 12.4/2.3 at compile time.

(iii) int k=2;

if(k) go to L3;

It is evaluated as :

go to L3 (Because k = 2 which implies condition is always true)

Loop Optimization

Code Motion / Frequency Reduction

```
a=100;
While(a>0)
{
    x=y+z;
    if(a%x == 0)
          printf("%d",x);
}
```

Code Motion / Frequency Reduction

```
a=100;
x=y+z;
While(a>0)
{
    if(a%x == 0)
          printf("%d",x);
}
```

Loop Fusion / Loop Jamming

```
int i, a[100], b[100];
for(i=0; i< 100;  i++)
      a[i]=1;
for(i=0; i< 100;  i++)
      b[i]=2;
```

Loop Fusion / Loop Jamming

```
int i, a[100], b[100];
for(i=0; i< 100;  i++)
{
      a[i]=1;
      b[i]=2;
}
```

Loop Unrolling

```
for(i=0; i< 3;  i++)
{
      printf("Sarita ");
}
```

Loop Unrolling

```
printf("Sarita Sarita Sarita");
```

Induction Variable & Strength Reduction

An induction variable is a variable used in a loop (for, while...). It controls the iteration of the loop.
Strength reduction is one of the techniques used to optimize loop induction variables.

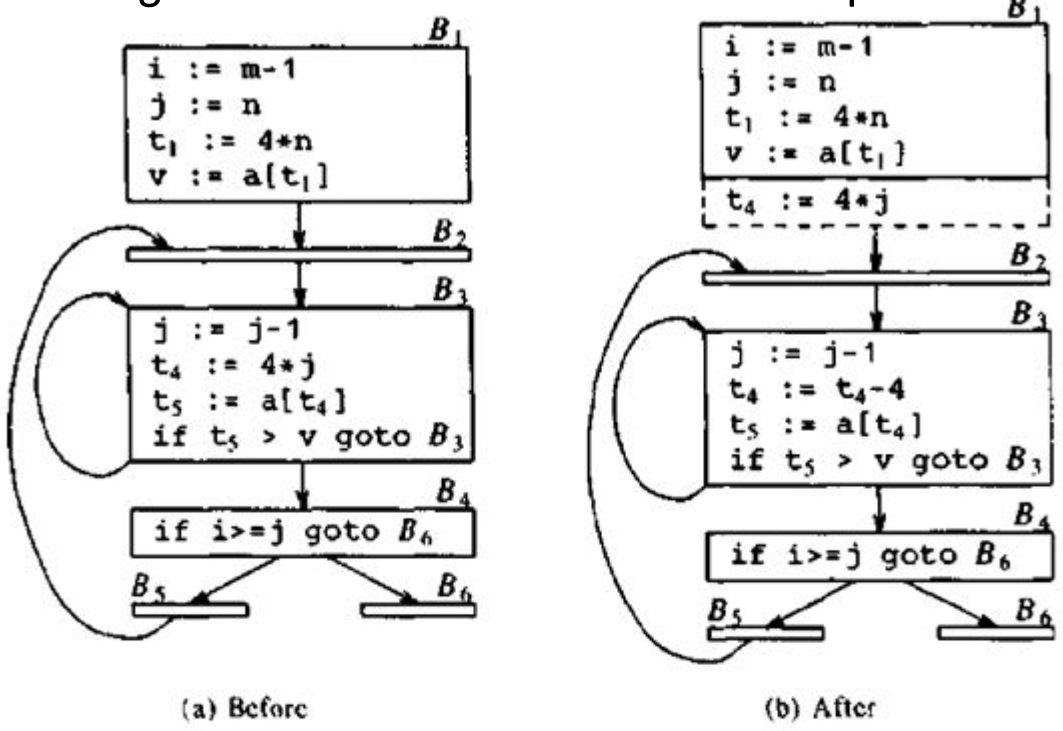


Fig. 10.9. Strength reduction applied to 4*j in block B_3.

```
// before strength reduction
int main()
{
        int a = 2;
        int b = a * 3;
}

// after strength reduction
int main()
{
        int a = 2;
        int b = a + a + a; // replaced * with three +
operations
}
```

Induction Variable & Strength Reduction

An induction variable is a variable used in a loop (for, while…). It controls the iteration of the loop.
Strength reduction is one of the techniques used to optimize loop induction variables.

```
// before strength reduction
int main()
{
        int a = 2;
        int b = a * 4;
}

// after strength reduction
int main()
{
        int a = 2;
        int b = (a << 2); // left shift by 2
}
```

```
// before strength reduction
int main()
{
        int a = 8;
        int b = a / 4;
}

// after strength reduction
int main()
{
        int a = 8;
        int b = (a >> 2); // right shift by 2
}
```

Induction Variable & Strength Reduction

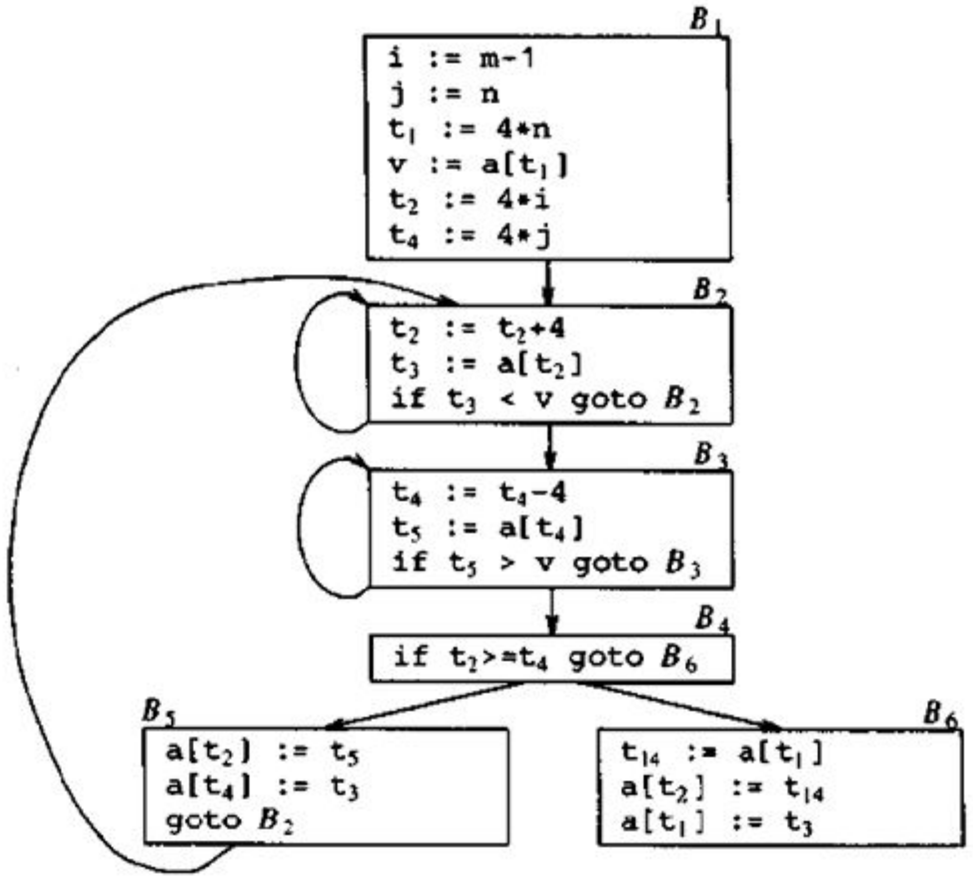
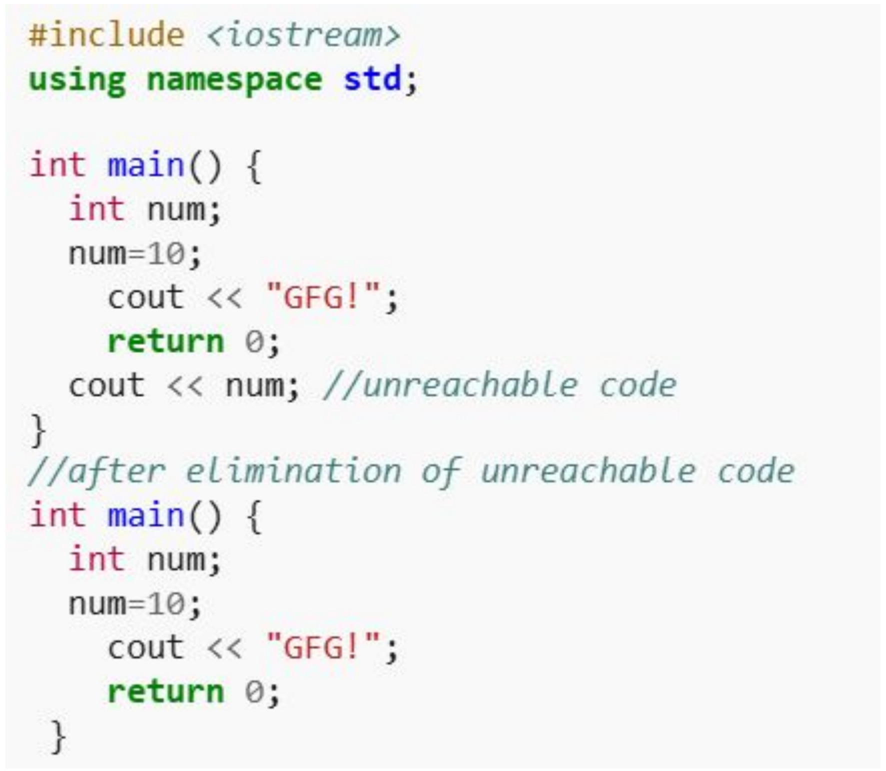


Fig. 10.10. Flow graph after induction-variable elimination.

Dead code Elimination

- Copy propagation often leads to making assignment statements into dead code.
- A variable is said to be dead if it is never used after its last definition.
- In order to find the dead variables, a data flow analysis should be done.



```cpp
#include <iostream>
using namespace std;

int main() {
  int num;
  num=10;
    cout << "GFG!";
    return 0;
  cout << num; //unreachable code
}
//after elimination of unreachable code
int main() {
  int num;
  num=10;
    cout << "GFG!";
    return 0;
 }
```